# Random Access to Grammar-Compressed Strings

Philip Bille * Gad M. Landau † Rajeev Raman ‡ Kunihiko Sadakane §
Srinivasa Rao Satti ¶ Oren Weimann ‖

**Abstract**

Let $S$ be a string of length $N$ compressed into a context-free grammar $\mathcal{S}$ of size $n$. We present two representations of $\mathcal{S}$ achieving $O(\log N)$ random access time, and either $O(n \cdot \alpha_k(n))$ construction time and space on the pointer machine model, or $O(n)$ construction time and space on the RAM. Here, $\alpha_k(n)$ is the inverse of the $k^{th}$ row of Ackermann's function. Our representations also efficiently support decompression of any substring in $S$: we can decompress any substring of length $m$ in the same complexity as a single random access query and additional $O(m)$ time. Combining these results with fast algorithms for uncompressed approximate string matching leads to several efficient algorithms for approximate string matching on grammar-compressed strings without decompression. For instance, we can find all approximate occurrences of a pattern $P$ with at most $k$ errors in time $O(n(\min\{|P|k, k^4 + |P|\} + \log N) + \text{occ})$, where occ is the number of occurrences of $P$ in $S$. Finally, we are able to generalize our results to navigation and other operations on grammar-compressed *trees*.

All of the above bounds significantly improve the currently best known results. To achieve these bounds, we introduce several new techniques and data structures of independent interest, including a predecessor data structure, two "biased" weighted ancestor data structures, and a compact representation of heavy-paths in grammars.

## 1 Introduction

Modern textual or semi-structured databases, e.g. for biological and WWW data, are huge, and are typically stored in compressed form. A query to such databases will typically retrieve only a small portion of the data. This presents several challenges: how to query the compressed data directly and efficiently, without the need for additional data structures (which can be many times larger than the compressed data), and how to retrieve the answers to the queries. In many practical cases, the naive approach of first decompressing the entire data and then processing it is completely unacceptable – for instance XML data compresses by an order of magnitude on disk [25] but *expands* by an order of magnitude when represented in-memory [22]; as we will shortly see, this approach is very problematic from an asymptotic perspective as well. Instead we want to support this functionality directly on the compressed data.

We focus on two data types, *strings* and *labelled trees*, and consider the former first. Let $S$ be a string of length $N$ from an alphabet $\Sigma$, given in a compressed representation $\mathcal{S}$ of size $n$. The *random access problem* is to compactly represent $\mathcal{S}$ while supporting fast random access queries, that is, given an index $i$, $1 \leq i \leq N$, report $S[i]$. More generally, we want to support *substring decompression*, that is, given a pair of indices $i$ and $j$, $1 \leq i \leq j \leq N$, report the substring $S[i] \cdots S[j]$. The goal is to use little space for the representation of $\mathcal{S}$ while supporting fast random access and substring decompression. Once we obtain an efficient substring decompression method, it can also serve as a basis for a compressed version of classical pattern matching. For example, given an (uncompressed) pattern string $P$ and $\mathcal{S}$, the *compressed pattern matching problem* is to find all occurrences of $P$ within $S$ more efficiently than to naively decompress $\mathcal{S}$ into $S$ and then search for $P$ in $S$. An important variant of the pattern matching problem is when we allow approximate matching (i.e., when $P$ is allowed to appear in $S$ with some errors).

We consider these problems in the context of *grammar-based compression*, where one replaces a long string by a small context-free grammar (CFG) that generates this string (and this string only, see Fig. 1(a)). Such grammars capture many popular

*DTU Informatics, Technical University of Denmark, Denmark. phbi@imm.dtu.dk

†Department of Computer Science, University of Haifa, Israel landau@cs.haifa.ac.il and Department of Computer Science and Engineering, NYU-Poly, USA. Partially supported by the National Science Foundation (award 0904246) Israel Science Foundation (grants 35/05, 347/09), the Israel-Korea Scientific Research Cooperation, Yahoo, and the United States-Israel Binational Science Foundation (grant 2008217).

‡Department of Computer Science, University of Leicester, UK. r.raman@mcs.le.ac.uk. Raman was supported by Royal Society Travel Grant TG091629.

§National Institute of Informatics, Japan. sada@nii.ac.jp

¶School of Computer Science and Engineering, Seoul National University, S. Korea. ssrao@cse.snu.ac.kr

‖Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Israel. oren.weimann@weizmann.ac.il



Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

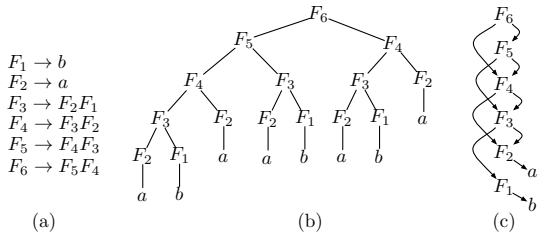



Figure 1: (a) A context-free grammar generating the string abaababa. (b) The corresponding parse tree. (c) The acyclic graph defined by the grammar.

compression schemes including the Lempel-Ziv family [58, 60, 61], Sequitur [50], Run-Length Encoding, Re-Pair [41], and many more [6–8, 28, 37, 38, 55, 59]. All of these are or can be transformed into equivalent grammar-based compression schemes with little expansion [17, 52]. In general, the size of the grammar, defined as the total number of symbols in all derivation rules, can be exponentially smaller than the string it generates. From an algorithmic perspective, the properties of compressed data were used to accelerate the solutions to classical problems on strings including exact pattern matching [4, 36, 42, 44, 56] and approximate pattern matching [3, 9, 12, 14, 16, 21, 33, 35, 36, 43, 49].

We also consider the problem of representing an ordered rooted tree $T$ (of arbitrary degree) with $N$ nodes, where each node is labelled with a character from $\Sigma$ (called *labelled* trees [25]). We assume that the tree $T$ is compressed by sharing identical subtrees, giving a DAG with $n$ nodes (see Fig. 6 in Section 8). This approach has recently been applied successfully to compress XML documents [13, 15] and [13] also note that this representation aids the matching of XPath patterns, but their algorithm partially decompresses the DAG. Indeed [15, p468] specifically mention the problem of navigating the XML tree without decompressing the DAG, and present algorithms whose running time is linear in the grammar size for randomly accessing the nodes of the tree.

**Our Results.** We present new representations of grammar-compressed strings and trees. We consider two models, the *pointer machine* [57] and the *word RAM* (henceforth just RAM) [31]. We further make the assumption that all memory cells can contain $\log N$-bit integers – this many bits are needed just to represent the input to a random access query. Let $\alpha_k(n)$ be the inverse of the $k^{th}$ row of Ackermann's function[1]. For strings, we show:

THEOREM 1.1. *For a CFG $\mathcal{S}$ of size $n$ representing a string of length $N$ we can decompress a substring of length $m$ in time $O(m + \log N)$*

*(i) after $O(n \cdot \alpha_k(n))$ preprocessing time and space for any fixed $k$, or,*

*(ii) after $O(n)$ preprocessing time and space on the RAM model.*

Next, we show how to combine Theorem 1.1 with any black-box (uncompressed) approximate string matching algorithm to solve the corresponding compressed approximate string matching problem over grammar-compressed strings. We obtain the following connection between classical (uncompressed) and grammar-compressed approximate string matching. Let $t(m)$ and $s(m)$ be the time and space bounds of some (uncompressed) approximate string matching algorithm on strings of lengths $O(m)$, and let occ be the number of occurrences of $P$ in $S$.

THEOREM 1.2. *Given a CFG $\mathcal{S}$ of size $n$ representing a string of length $N$ and a string $P$ of length $m$ we can find all approximate occurrences of $P$ in $\mathcal{S}$ in time $O(n(m + t(m)) + \log N) + \text{occ})$ and*

*(i) in space $O(n \cdot \alpha_k(n) + m + s(m))$ on the pointer machine model and*

*(ii) in space $O(n + m + s(m) + \text{occ})$ on the RAM model.*

Coming to the tree representation problem, suppose that nodes of the uncompressed tree $T$ are numbered $1, \ldots, N$ in pre-order. We mainly consider the operation access($i$), which returns the symbol associated with node $i$, and also consider a number of "navigation" operations, including parent($i$) and lca($i, j$), which return the number of the node that the parent of $i$ or the LCA of $i$ and $j$, respectively (a full list of navigation operations can be found in Section 8). We show:

THEOREM 1.3. *Given a compressed DAG with $m$ edges and $n$ nodes that represents a labelled tree with $N$ nodes, we can support* access, *as well as navigation, in $O(\log N)$ time using:*

[1]The inverse Ackermann function $\alpha_k(n)$ can be defined by $\alpha_k(n) = 1 + \alpha_k(\alpha_{k-1}(n))$ so that $\alpha_1(n) = n/2$, $\alpha_2(n) = \log n$, $\alpha_3(n) = \log^* n$, $\alpha_4(n) = \log^{**} n$ and so on. Here, $\log^{**} n$ is the number of times the $\log^*$ function is applied to $n$ to produce a constant.


Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

*(i)* $O(m\alpha_k(m))$ *words and preprocessing time on the pointer machine model;*

*(ii)* $O(m)$ *words and preprocessing time on the RAM model.*

**Related Work.** We now describe how our work relates to existing results. We assume without loss of generality that the grammars are in fact *straight-line programs* (SLPs) and so on the righthand side of each grammar rule there are either exactly two variables or one terminal symbol.

**The Random Access Problem.** If we use $O(N)$ space we can access any character in constant time by storing $S$ explicitly in an array. Alternatively, we can compute and store the sizes of strings derived by each grammar symbol in $\mathcal{S}$. This only requires $O(n)$ space and allows to simulate a top-down search expanding the grammar's derivation tree in constant time per node. Consequently, a random access takes time $O(h)$, where $h$ is the height of the derivation tree and can be as large as $\Omega(n)$. Although any SLP of size $n$ generating a string of length $N$ can be converted into an SLP with derivation tree height $O(\log N)$ [17, 52], the size of the SLP increases to $O(n \log N)$. Thus, the simple top-down traversal either has poor worst-case performance or uses non-linear space. Surprisingly, the only known improvement to the simple top-down traversal is a recent succinct representation of grammars, due to Claude and Navarro [19]. They reduce the space from $O(n \log N)$ *bits* to $O(n \log n) + n \log N$ bits at the cost of increasing the query time to $O(h \log n)$.

**The Substring Decompression Problem.** Using the simple random access trade-off we get an $O(n)$ space solution that supports substring decompression in $O(hm)$ time. Gasieniec et al. [29, 30] showed how to improve the decompression time to $O(h+m)$ while maintaining $O(n)$ space. Also, the representation of [19] supports substring decompression in time $O((h+m)\log n)$.

**The Compressed Pattern Matching Problem.** In approximate pattern matching, we are given two strings $P$ and $S$ and an *error threshold* $k$. The goal is to find all ending positions of substrings of $S$ that are "within distance $k$" of $P$ under some metric, e.g. the *edit distance* metric, where the distance is the number of edit operations needed to convert one substring to the other.

In classical (uncompressed) approximate pattern matching, a simple algorithm [54] solves this problem (under edit distance) in $O(Nm)$ time and $O(m)$ space, where $N$ and $m$ are the lengths of $S$ and $P$ respectively. Several improvements of this result are known (see e.g. [48]). Two well-known improvements for small values of $k$ are the $O(Nk)$ time algorithm of Landau and Vishkin [40] and the $O(Nk^4/m + N)$ time algorithm of Cole and Hariharan [20]. Both of these can be implemented in $O(m)$ space. The use of compression led to many speedups using various compression schemes [3, 9, 12, 14, 16, 21, 33, 35, 36, 43, 49]. The most closely related to our work is approximate pattern matching for LZ78 and LZW compressed strings [12,35,49], which can be solved in time $O(n(\min\{mk, k^4 + m\}) + \text{occ})$ [12], where $n$ is the compressed length under the LZ compression.

Theorem 1.2 gives us the first non-trivial algorithms for approximate pattern matching over any grammar compressed string. For instance, if we plug in the Landau-Vishkin [40] or Cole-Hariharan [20] algorithms in Theorem 1.2(i) we obtain an algorithm with $O(n(\min\{mk, k^4+m\} + \log N) + \text{occ})$ time and $O(n \cdot \alpha_k(n) + m + \text{occ})$ space. Note that any algorithm (not only the above two) and any distance metric (not only edit distance) can be applied to Theorem 1.2. For example, under the Hamming distance measure we can combine our algorithm with a fast algorithm for the (uncompressed) approximate string matching problem for the Hamming distance measure [5].

**Overview.** Before diving into technical details, we give an outline of the paper and of the new techniques and data structures that we introduce and believe to be of independent interest. We first focus on the string random acccess problem. Let $\mathcal{S}$ be a SLP of size $n$ representing a string of length $N$. We begin in Section 2 by defining a forest $H$ of size $n$ that represents the heavy paths [32] in the parse tree of $\mathcal{S}$. We then combine the forest $H$ with an existing *weighted ancestor* data structure[2], leading to a first solution with $O(\log N \log\log N)$ access time and linear space (Lemma 2.1). The main part of the paper focuses on reducing the random access time to $O(\log N)$.

In Section 3, we observe that it is better to replace the doubly-logarithmic weighted ancestor search in Lemma 2.1 by a (logarithmic) *biased* ancestor search. In a biased search, we want to find the predecessor of a given integer $p$ in a set of integers

[2] A weighted ancestor query $(v, p)$ asks for the lowest ancestor of $v$ whose weighted distance from $v$ is at least $p$.





Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

$0 = l_0 < l_1 < \ldots < l_k = U$, in $O(\log(U/x))$ time, where $x = |\text{successor}(p) - \text{predecessor}(p)|$.[3]. Using biased search, the $O(\log N)$ predecessor queries on $H$ add up to just $O(\log N)$ time overall. Our main technical contribution is to design two new space-efficient data structures that perform biased searches on sets defined by any path from a node $u \in H$ to the root of $u$'s tree. In Section 3 we describe the central building block of the first data structure – the *interval-biased search tree*, which is a new, simple linear-time constructible, linear space, biased search data structure. We cannot directly use this data structure on every node-to-root path in $H$, since that would take $O(n^2)$ preprocessing time and space. In Section 4 we first apply a heavy path decomposition to $H$ itself and navigate between these paths using weighted ancestor queries on a related tree $L$. This reduces the preprocessing time to $O(n \log n)$. To further reduce the preprocessing, we partition $L$ into disjoint trees in the spirit of Alstrup et al. [2]. One of these trees has $O(n/\log n)$ leaves and can be pre-processed using the solution above. The other trees all have $O(\log n)$ leaves and we handle them recursively. However, before we can recurse on these trees they are modified so that each has $O(\log n)$ vertices (rather than leaves). This is done by another type of path decomposition (i.e. not a heavy-path decomposition) of $L$. By carefully choosing the sizes of the recursive problems we get Theorem 1.1(i) (for the case $m = 1$).

For the RAM model, in Section 5, we generalize *biased skip lists* [10] to *biased skip trees*, where every path from a node $u \in H$ to $u$'s root is a biased skip list, giving the required time complexity. While a biased skip list takes linear space [34], a biased skip tree may have $\Omega(n \log N)$ pointers and hence non-linear space, since in a biased skip list, "overgrown" nodes (those with many more pointers than justified by their weight) are amortized over those ancestors which have an appropriate number of pointers. When used in $H$, however, the parent of an "overgrown" node may have many "overgrown" children, all sharing the same set of ancestors, and the amortization fails. We note that no node will have more than $O(\log N)$ pointers, and use a sequence of $O(\log N)$ *succinct* trees [47] of $O(|H|) = O(n)$ bits each to represent the skip list pointers, using $O(n \log N)$ bits or $O(n)$ words in all. These succinct trees support in $O(1)$ time a new *coloured ancestor query* – a natural operation that may find other uses – using which we are able to follow skip list pointers in $O(1)$ time, giving the bounds of Theorem 1.1(ii) (for the case $m = 1$).

We extend both random access solutions to the substring decompression in Section 6, and in Section 7 we combine our substring decompression result with a technique of [12] to obtain an algorithm for approximate matching grammar-compressed strings (giving the bounds of Theorem 1.2). The algorithm computes the approximate occurrences of the pattern in a single bottom-up traversal of the grammar. At each step we use the substring decompression algorithm to decode a relevant small portion of string, thus avoiding a full decompression.

Finally, in Section 8, we describe the differences between the random access operation in trees and that in strings.

## 2 Fast Random Access in Linear Space

In the rest of the paper, we let $\mathcal{S}$ denote an SLP of size $n$ representing a string of length $N$, and let $T$ be the corresponding parse tree (see Fig. 1(b)). In this section we present an $O(n)$ space representation of $\mathcal{S}$ that supports random access in $O(\log N \log\log N)$ time, which also introduces the general framework. To achieve this we partition $\mathcal{S}$ into disjoint paths according to a *heavy path decomposition* [32], and from these form the *heavy path forest*, which is of size $O(n)$.

**Heavy Path Decompositions.** Similar to Harel and Tarjan [32], we define the *heavy path decomposition* of the parse tree $T$ as follows. For each node $v$ define $T(v)$ to be the subtree rooted at $v$ and let $\text{size}(v)$ be the number of descendant leaves of $v$. We classify each node in $T$ as either *heavy* or *light* based upon $\text{size}(v)$.[4] The root is light. For each internal node $v$ we pick a child of maximum size and classify it as heavy. The heavy child of $v$ is denoted $\text{heavy}(v)$. The remaining children are light. An edge to a light child is a *light edge* and an edge to a heavy child is a *heavy edge*. Removing the light edges we partition $T$ into *heavy paths*. A *heavy path suffix* is a simple path $v_1, \ldots, v_k$ from a node $v_1$ to a leaf in $T(v_1)$, such that $v_{i+1} = \text{heavy}(v_i)$, for $i = 1, \ldots, k-1$. If $u$ is a light child of $v$ then $\text{size}(u) \leq \text{size}(v)/2$ since otherwise $u$ would be heavy. Consequently, the number of light

[3] Note that we need a slightly different property than so-called *optimum* binary search trees [39, 45] – we do not want to minimize the total external path length but rather ensure that *each* item is at its ideal depth as in [11]

[4] Note that our definition of heavy paths is slightly different than the usual one. We construct our heavy paths according to the number of leaves of the subtrees and not the total number nodes.


Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

edges on a path from the root to a leaf is at most $O(\log N)$ [32].

We extend heavy path decomposition of trees to SLPs in a straightforward manner. We consider each grammar variable $v$ as a node in the directed acyclic graph defined by the grammar (see Fig. 1(c)). For a node $v$ in $\mathcal{S}$ let $S(v)$ be the substring induced by the parse tree rooted at $v$ and define the size of $v$ to be the length of $S(v)$. We define the heavy paths in $\mathcal{S}$ as in $T$ from the size of each node. Since the size of a node $v$ in $\mathcal{S}$ is the number of leaves in $T(v)$ the heavy paths are well-defined and we may reuse all of the terminology for trees on SLPs. In a single $O(n)$ time bottom-up traversal of $\mathcal{S}$ we can compute the sizes of all nodes and hence the heavy path decomposition of $\mathcal{S}$.

**Fast Random Access in Linear Space.** Our data structure represents the following information for each heavy path suffix $v_1, \ldots, v_k$ in $\mathcal{S}$.

- The length $\text{size}(v_1)$ of the string $S(v_1)$.
- The index $z$ of $v_k$ in the left-to-right order of the leaves in $T(v_1)$ and the character $S(v_1)[z]$.
- A predecessor data structure for the *left size sequence* $l_0, l_1, \ldots, l_k$, where $l_i$ is the sum of 1 plus the sizes of the left and light children of the first $i$ nodes in the heavy path suffix.
- A predecessor data structure for the *right size sequence* $r_0, \ldots, r_k$, where $r_i$ is the sum of 1 plus the sizes of the right and light children of the first $i$ nodes in the heavy path suffix.

With this information we perform a top down search of $T$ as follows. Suppose that we have reached node $v_1$ with heavy path suffix $v_1, \ldots, v_k$ and our goal is to access the character $S(v_1)[p]$. We then compare $p$ with the index $z$ of $v_k$. There are three cases (see Fig. 2 for an example):

1. If $p = z$ we report the stored character $S(v_1)[z]$ and end the search.
2. If $p < z$ we compute the predecessor $l_i$ of $p$ in the left size sequence. We continue the top down search from the left child $u$ of $v_{i+1}$. The position of $p$ in $T(u)$ is $p - l_i + 1$.
3. If $p > z$ we compute the predecessor $r_i$ of $\text{size}(v_1) - p$ in the right size sequence. We continue the top down search from the right child $u$ of $v_{i+1}$. The position of $p$ in $T(u)$ is $p - (z + \sum_{j=i+2}^{k} \text{size}(v_j))$ (note that we can compute the sum in constant time as $r_k - r_{i+2}$).

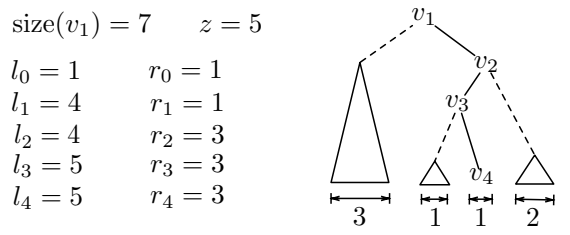



Figure 2: Ancestor search in $H$. The left and right size sequences for a heavy path suffix $v_1, v_2, v_3, v_4$. The dotted edges are to light subtrees and the numbers in the bottom are subtree sizes. A search for $p = 5$ returns the stored character for $S(v_1)[z]$. A search for $p = 4$ computes the predecessor $l_2$ of 4 in the left size sequence. The search continues in the left subtree of $v_3$ for position $p - l_2 + 1 = 4 - 4 + 1 = 1$. A search for $p = 6$ computes the predecessor $r_1$ of $7 - 6 = 1$ in the right size sequence. The search continues in the right subtree of $v_2$ for position $p - z = 6 - 5 = 1$.

The total length of all heavy path suffixes is $O(n^2)$, thus making it unattractive to treat each suffix independently. We show how to compactly represent all of the predecessor data structures from the algorithm of the previous section in $O(n)$ space, and introduce the *heavy path suffix forest* $H$ of $\mathcal{S}$. The nodes of $H$ are the nodes of $S$ and a node $u$ is the parent of $v$ in $H$ iff $u$ is the heavy child of $v$ in $\mathcal{S}$. Thus, a heavy path suffix $v_1, \ldots, v_k$ in $\mathcal{S}$ is a sequence of ancestors from $v_1$ in $H$. We label the edge from $v$ to its parent $u$ by a left weight and right weight defined as follows. If $u$ is the left child of $v$ in $\mathcal{S}$ the left weight is 0 and the right weight is $\text{size}(v')$ where $v'$ is the right child of $v$. Otherwise, the right weight is 0 and the left weight is $\text{size}(v')$ where $v'$ is the left child of $v$. Heavy path suffixes in $\mathcal{S}$ consist of unique nodes and therefore $H$ is a forest. A heavy path suffix in $\mathcal{S}$ ends at one of $|\Sigma|$ leaves in $\mathcal{S}$ and therefore $H$ consists of $|\Sigma|$ trees each rooted at a unique character of $\Sigma$. The total size of $H$ is $O(n)$ and we may easily compute it from the heavy path decomposition of $\mathcal{S}$ in $O(n)$ time.

A predecessor query on a left size sequence and right size sequence of a heavy path suffix $v_1, \ldots, v_k$ is now equivalent to a *weighted ancestor query* on the left weights and right weights of $H$, respectively. Farach-Colton and Muthukrishnan [24] showed how





Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

to support weighted ancestor queries in $O(\log \log N)$ time after $O(n)$ space and preprocessing time. Hence, if we plug this in to our algorithm we obtain $O(\log N \log \log N)$ query time with $O(n)$ preprocessing time and space. In summary, we have the following result.

LEMMA 2.1. *For an SLP $\mathcal{S}$ of size $n$ representing a string of length $N$ we can support random access in time $O(\log N \log \log N)$ after $O(n)$ preprocessing time and space.*

## 3 Interval-Biased Search Trees

In this section we reduce the $O(\log N \log \log N)$ random access time on an SLP $\mathcal{S}$ in Lemma 2.1 to $O(\log N)$. Recall that $O(\log N \log \log N)$ was a result of performing $O(\log N)$ predecessor($p$) queries, each in $O(\log \log N)$ time. In this section, we introduce a new predecessor data structure – the *interval-biased search tree*. Each predecessor($p$) query on this data structure requires $O(\log \frac{U}{x})$ time, where $x$ = successor($p$) – predecessor($p$), and $U$ is the universe.

To see the advantage of $O(\log \frac{U}{x})$ predecessor queries over $O(\log \log N)$, suppose that after performing the predecessor query on the first heavy path of $T$ we discover that the next heavy path to search is the heavy path suffix originating in node $u$. This means that the first predecessor query takes $O(\log \frac{N}{|S(u)|})$ time. Furthermore, the elements in $u$'s left size sequence (or right size sequence) are all from a universe $\{0, 1, \ldots, |S(u)|\}$. Therefore, the second predecessor query takes $O(\log \frac{|S(u)|}{x})$ where $x = |S(u')|$ for some node $u'$ in $T(u)$. The first two predecessor queries thus require time $O(\log \frac{N}{|S(u)|} + \log \frac{|S(u)|}{x}) = O(\log \frac{N}{x})$. The time required for all $O(\log N)$ predecessor queries telescopes similarly for a total of $O(\log N)$.

We next show how to construct an interval-biased search tree in linear time and space. Simply using this tree on each heavy path suffix of $\mathcal{S}$ already results in the following lemma.

LEMMA 3.1. *For an SLP $\mathcal{S}$ of size $n$ representing a string of length $N$ we can support random access in time $O(\log N)$ after $O(n^2)$ preprocessing time and space.*

**A Description of the Tree.** We now define the interval-biased search tree associated with $\hat{n}$ integers $l_1 \leq \ldots \leq l_{\hat{n}}$ from a universe $\{0, 1, \ldots, \hat{N}\}$. For simplicity, we add the elements $l_0 = 0$ and $l_{\hat{n}+1} = \hat{N}$. The interval-biased search tree is a binary tree that stores the intervals $[l_0, l_1], [l_1, l_2], \ldots, [l_{\hat{n}}, l_{\hat{n}+1}]$ with a single interval in each node. The tree is described recursively:

1. Let $i$ be such that $(l_{\hat{n}+1} - l_0)/2 \in [l_i, l_{i+1}]$. The root of the tree stores the interval $[l_i, l_{i+1}]$.

2. The left child of the root is the interval-biased search tree storing the intervals $[l_0, l_1], \ldots, [l_{i-1}, l_i]$, and the right child is the interval-biased search tree storing the intervals $[l_{i+1}, l_{i+2}], \ldots, [l_{\hat{n}}, l_{\hat{n}+1}]$.

When we search the tree for a query $p$ and reach a node corresponding to the interval $[l_i, l_{i+1}]$, we compare $p$ with $l_i$ and $l_{i+1}$. If $l_i \leq p \leq l_{i+1}$ then we return $l_i$ as the predecessor. If $p < l_i$ (resp. $p > l_{i+1}$) we continue the search in the left child (resp. right child). Notice that an interval $[l_i, l_{i+1}]$ of length $x = l_{i+1} - l_i$ such that $\hat{N}/2^{j-1} \leq x \leq \hat{N}/2^j$ is stored in a node of depth at most $j$. Therefore, a query $p$ whose predecessor is $l_i$ (and whose successor is $l_{i+1}$) terminates at a node of depth at most $j$. The query time is thus $j \leq 1 + \log \frac{\hat{N}}{x} = O(\log \frac{\hat{N}}{x})$ which is exactly what we desire as $x$ = successor($p$) – predecessor($p$). We now give an $O(\hat{n})$ time and space algorithm for constructing the tree.

**A Linear-Time Construction of the Tree.** We describe an $O(\hat{n})$ time and space top-down construction of the interval-biased search tree storing the intervals $[l_j, l_{j+1}], \ldots, [l_k, l_{k+1}]$. We focus on finding the interval $[l_i, l_{i+1}]$ to be stored in its root. The rest of the tree is constructed recursively so that the left child is a tree storing the intervals $[l_j, l_{j+1}], \ldots, [l_{i-1}, l_i]$ and the right child is a tree storing the intervals $[l_{i+1}, l_{i+2}], \ldots, [l_k, l_{k+1}]$.

We are looking for an interval $[l_i, l_{i+1}]$ such that $i$ is the largest value where $l_i \leq (l_{k+1} + l_j)/2$ holds. We can find this interval in $O(\log(k - j))$ time by doing a binary search for $(l_{k+1} + l_j)/2$ in the subarray $l_j, l_{j+1}, \ldots, l_{k+1}$. However, notice that we are not guaranteed that $[l_i, l_{i+1}]$ partitions the intervals in the middle. In other words, $i - j$ can be much larger than $k - i$ and vice versa. This means that the total time complexity of all the binary searches we do while constructing the entire tree can amount to $O(n \log n)$ and we want $O(n)$. To overcome this, notice that we can find $[l_i, l_{i+1}]$ in $\min\{\log(i - j), \log(k - i)\}$ time if we use a doubling search from both sides of the subarray. That is, if prior to the binary search, we narrow the search space by doing a parallel scan of the elements $l_j, l_{j+2}, l_{j+4}, l_{j+8}, \ldots$ and $l_k, l_{k-2}, l_{k-4}, l_{k-8}, \ldots$. This turns out to be crucial


Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

for achieving $O(n)$ total construction time as we now show.

To verify the total construction time, we need to bound the total time required for all the binary searches. Let $T(\hat{n})$ denote the time complexity of all the binary searches, then $T(\hat{n}) = T(i) + T(\hat{n} - i) + \min\{\log i, \log(\hat{n} - i)\}$ for some $i$. Setting $d = \min\{i, \hat{n} - i\} \leq \hat{n}/2$ we get that $T(\hat{n}) = T(d) + T(\hat{n} - d) + \log d$ for some $d \leq \hat{n}/2$, which is equal[5] to $O(\hat{n})$.

**Final Tuning.** We need one last important property of the interval-biased search tree[6]. Suppose that right before doing a predecessor($p$) query we know that $p > l_k$ for some $k$. We can reduce the query time to $O(\log \frac{\hat{N} - l_k}{x})$ by computing for each node its lowest common ancestor with the node $[l_{\hat{n}}, l_{\hat{n}+1}]$, in a single traversal of the tree. Then, when searching for $p$, we can start the search in the lowest common ancestor of $[l_k, l_{k+1}]$ and $[l_{\hat{n}}, l_{\hat{n}+1}]$ in the interval-biased search tree.

## 4 Closing the Time-Space Tradeoffs for Random Access

In this section we will use the interval-biased search tree to achieve $O(\log N)$ random access time but near-linear space usage and preprocessing time (instead of $O(n^2)$ as in Lemma 2.1). We design a novel *weighted ancestor* data structure on $H$ via a heavy path decomposition of $H$ itself. We use interval-biased search trees for each heavy path $P$ in this decomposition: one each for the left and right size sequences. It is easy to see that the total size of all these interval-biased search trees is $O(n)$. We focus on queries of the left size sequence, the right size sequence is handled similarly.

Let $P$ be a heavy path in the decomposition, let $v$ be a vertex on this path, and let $w(v, v')$ be the weight of the edge between $v$ and his child $v'$, We denote by $b(v)$ the weight of the part of $P$ below $v$ and by $t(v)$ the weight above $v$. As an example, consider the green heavy path $P = (v_5\text{-}v_4\text{-}v_8\text{-}v_9)$ in Fig. 3, then $b(v_4) = w(v_4, v_8) + w(v_8, v_9)$ and $t(v_4) = w(v_5, v_4)$. In general, if $P = (v_k\text{-}v_{k-1}\text{-}\cdots\text{-}v_1)$ then $v_1$ is a leaf in $H$ and $b(v_{i+1})$ is the $i$'th element in $P$'s predecessor data structure. The $b(\cdot)$ and $t(\cdot)$ values of all vertices can easily be computed in $O(n)$ time.

Recall that given any vertex $u$ in $H$ and any $0 \leq p \leq N$ we need to be able to find the lowest ancestor $v$ of $u$ whose weighted distance from $u$ is at least $p$. If we want the total random access time to be $O(\log N)$ then finding $v$ should be done in $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ time where $v'$ is the child of $v$ which is also an ancestor of $u$. If both $u$ and $v$ are on the same heavy path $P$ in the decomposition, a single predecessor($p'$) query on $P$ would indeed find $v$ in $O(\log \frac{t(u)}{w(v,v')}) = O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ time, where $p' = p + b(u)$. This follows from the property we described at the end of Section 3.

The problem is thus to locate $v$ when, in the decomposition of $H$, $v$ is on the heavy path $P$ but $u$ is not. To do so, we first locate a vertex $w$ that is both an ancestor of $u$ and belongs to $P$. Once $w$ is found, if its weighted distance from $u$ is greater than $p$ then $v = w$. Otherwise, a single predecessor($p''$) query on $P$ finds $v$ in $O(\log \frac{t(w)}{w(v,v')})$ time, which is $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ since $t(w) \leq |S(u)|$. Here, $p'' = p$ - weight(path from $u$ to $w$ in $H$) + $b(w)$. We are therefore only left with the problem of finding $w$ and the weight of the path from $u$ to $w$.

**A Light Representation of Heavy Paths.** In order to navigate from $u$ up to $w$ we introduce the *light representation* $L$ of $H$. Intuitively, $L$ is a (non-binary) tree that captures the light edges in the heavy-path decomposition of $H$. Every path $P$ in the decomposition of $H$ corresponds to a single vertex $P$ in $L$, and every light edge in the decomposition of $H$ corresponds to an edge in $L$. If a light edge $e$ in $H$ connects a vertex $w$ with its child then the weight of the corresponding edge in $L$ is the original weight of $e$ plus $t(w)$. (See the edge of weight $w(v_4, v_3)+w(v_5, v_4)$ in Fig. 3).

The problem of locating $w$ in $H$ now translates to a weighted ancestor query on $L$. Indeed, if $u$ belongs to a heavy-path $P'$ then $P'$ is also a vertex in $L$ and locating $w$ translates to finding the lowest ancestor of $P'$ in $L$ whose weighted distance from $P'$ is at least $p - t(u)$. As a weighted ancestor data structure on $L$ would be too costly, we utilize the important fact that the height of $L$ is only $O(\log n)$ – the edges of $L$ correspond to light edges of $H$ – and construct, for every root-to-leaf path in $L$, an interval-biased search tree as its predecessor data structure. The total time and space for constructing these data structures is $O(n \log n)$. A query for finding the ancestor of $P'$ in $L$ whose weighted distance from $P'$ is at least

[5] By an inductive assumption that $T(\hat{n}) < 2\hat{n} - \log \hat{n} - 2$ we get that $T(\hat{n})$ is at most $2d - \log d - 2 + 2(\hat{n} - d) - \log(\hat{n} - d) - 2 + \log d = 2\hat{n} - \log(\hat{n} - d) - 4$, which is at most $2\hat{n} - \log \hat{n} - 3$ since $d \leq \hat{n}/2$.

[6] In fact, there exist linear-time-constructable predecessor data structures with query complexity only $O(\log \log \frac{\hat{N}}{x})$ [51]. They are more complicated than our tree, but more importantly, their query time cannot handle $\hat{N}$ reducing to $\hat{N} - l_k$.



Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

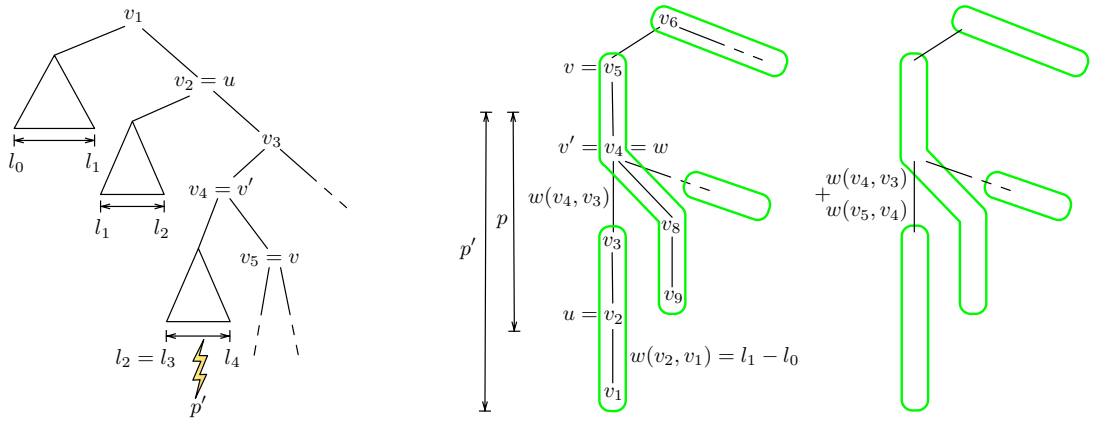



Figure 3: The parse tree $T$ of an SLP (left), the *heavy path suffix forest* $H$ (middle), and the *light representation* $L$ of $H$ (right). The heavy path decomposition of $H$ is marked (in green) and defines the vertex set of $L$.

$p - t(u)$ can then be done in $O(\log \frac{|S(u)|}{t(w)})$ time. This is $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ as $w(v, v') \leq t(w)$. We summarize this with the following lemma.

LEMMA 4.1. *For an SLP $\mathcal{S}$ of size $n$ representing a string of length $N$ we can support random access in time $O(\log N)$ after $O(n \log n)$ preprocessing time and space.*

As noted in the Introduction, the further reduction to $O(n\alpha_k(n))$ space and preprocessing time is achieved through a further decomposition of $L$. Intuitively, we partition $L$ into disjoint trees in the spirit of Alstrup et al. [2]. One of these trees has $O(n/\log n)$ leaves and can be pre-processed using the solution above. The other trees all have $O(\log n)$ leaves and we want to handle them recursively. However, for the recursion to work we will need to modify these trees so that each has $O(\log n)$ vertices (rather than leaves). As described in the following subsection, this is done by another type of path decomposition – a *branching decomposition*.

**An Inverse-Ackerman Type Bound.** We have just seen that after $O(n \log n)$ preprocessing we can support random access in $O(\log N)$ time. This superlinear preprocessing originates in the $O(n \log n)$-sized data structure that we construct on $L$ for $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$-time weighted ancestor queries. We now turn to reducing the preprocessing to be arbitrarily close to linear by recursively shrinking the size of this weighted ancestor data structure on $L$.

In order to do so, we perform a decomposition of $L$ that was originally introduced by Alstrup, Husfeldt, and Rauhe [2] for solving the the *marked ancestor* problem: Given the rooted tree $L$ of $n$ nodes, for every maximally high node whose subtree contains no more than $\log n$ *leaves*, we designate the subtree rooted at this node a *bottom tree*. Nodes not in a bottom tree make up the *top tree*. It is easy to show that the top tree has at most $n/\log n$ leaves and that this decomposition can be done in linear time.

Notice that we can afford to construct, for every root-to-leaf path *in the top tree*, an interval-biased search tree as its predecessor data structure. This is because there will be only $n/\log n$ such data structures and each is of size $\text{height}(L) = O(\log n)$. In this way, a weighted ancestor query that originates in a top tree node takes $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ time as required. The problem is therefore handling queries originating in bottom trees.

To handle such queries, we would like to recursively apply our $O(n \log n)$ weighted ancestor data structure on each one of the bottom trees. This would work nicely if the number of *nodes* in a bottom tree was $O(\log n)$. Unfortunately, we only know this about the number of its *leaves*. We therefore use a *branching representation* $B$ for each bottom tree. The number of *nodes* in the representation $B$ is indeed $\log n$ and it is defined as follows.

We partition a bottom tree into disjoint paths according to the following rule: A node $v$ belongs to the same path as its child unless $v$ is a branching-node (has more than one child). We associate each path $P$ in this decomposition with a unique interval-biased search tree as its predecessor's data structure. The *branching representation* $B$ is defined as follows.


Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

Every path $P$ corresponds to a single node in $B$. An edge $e$ connecting path $P'$ with its parent-path $P$ corresponds to an edge in $B$ whose weight is $e$'s original weight plus the total weighted length of the path $P'$ (See Fig. 4).

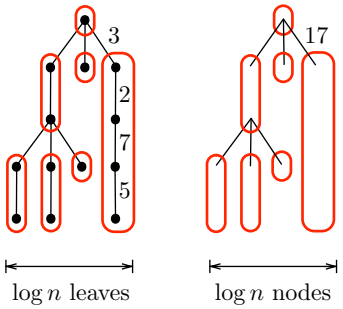



Figure 4: A bottom tree and its branching representation $B$. On the left is some *bottom tree* – a weighted tree with $\log n$ leaves. The bottom tree can be decomposed into $\log n$ paths (marked in red) each with at most one branching node. Replacing each such path with a single node we get the *branching representation* $B$ as depicted on the right. The edge-weight 17 is obtained by the original weight 3 plus the weighted path 2+7+5.

Each internal node in $B$ has at least two children and therefore the number of nodes in $B$ is $O(\log n)$. Furthermore, similarly to Section 4, our only remaining problem is weighted ancestor queries on $B$. Once the correct node is found in $B$, we can query the interval-biased search tree of its corresponding path in $L$ in $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ time as required.

Now that we can capture a bottom tree with its branching representation $B$ of logarithmic size, we could simply use our $O(n \log n)$ weighted ancestor data structure on every $B$. This would require an $O(\log n \log\log n)$-time construction for each one of the $n/\log n$ bottom trees for a total of $O(n \log\log n)$ construction time. In addition, every bottom tree node $v$ stores its weighted distance $d(v)$ from the root of its bottom tree. After this preprocessing, upon query $v$, we first check $d(v)$ to see whether the target node is in the bottom tree or the top tree. Then, a single predecessor query on the (bottom or top) tree takes $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ time as required.

It follows that we can now support random access on an SLP in time $O(\log N)$ after only $O(n \log\log n)$ preprocessing. In a similar manner we can use this $O(n \log\log n)$ preprocessing recursively on every $B$ to obtain an $O(n \log\log\log n)$ solution. Consequently, we can reduce the preprocessing to $O(n \log^* n)$ while maintaining $O(\log N)$ random access. Notice that if we do this naively then the query time increases by a $\log^* n$ factor due to the $\log^* n$ $d(v)$ values we have to check. To avoid this, we simply use an interval-biased search tree for every root-to-leaf path of $\log^* n$ $d(v)$ values. This only requires an additional $O(n \log^* n)$ preprocessing and the entire query remains $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$.

Finally, we note that choosing the recursive sizes more carefully (in the spirit of [1,18]) can reduce the $\log^* n$ factor down to $\alpha_k(n)$ for any fixed $k$. This gives Theorem 4.1:

THEOREM 4.1. *For an SLP $\mathcal{S}$ of size $n$ representing a string of length $N$ we can support random access in time $O(\log N)$ after $O(n \cdot \alpha_k(n))$ preprocessing time and space for any fixed $k$ on the pointer machine model.*

## 5 Biased Skip Trees

In this section we give an alternate representation of the heavy path forest $H$, that supports the "biased" predecessor search of the biased interval search tree; the space and preprocessing are both $O(n)$, but the data structure uses the more powerful word RAM model with word size $O(\log N)$ bits. For convenience of description, the predecessor search is expressed a little differently: suppose that we aim to access the $p$-th symbol of $S(v)$ for some node $v$, and suppose that $u$ is an ancestor of $v$ in $H$ (i.e. $u$ is a heavy descendant of $v$ in the parse tree); assume as previously that the desired symbol is not the symbol associated with the root of the tree in which $v$ is. We say that a *test* at $u$ is "true" if the desired symbol is in $u$'s heavy child, and "false" otherwise; this test is perfomed in $O(1)$ time by storing $l$ and $r$ values as before. Our objective is to find the lowest ancestor $u$ in $H$ of $v$ such that the test at $u$ is "false"; this search should take $O(\log(W_v/w_u) + 1)$ time, where for all nodes $u \in H$, $w_u = \text{size}(u')$, where $u'$ is the light child of $u$, and $W_u = \text{size}(u)$.

Our solution uses a static version of biased skip lists [10], generalized to trees. The initial objective is to assign a non-negative integral *color* $c_v$ to each node in $v \in H$ and there is a (logical) uni-directional linked list that points up the tree, such that all nodes on a leaf-to-root path whose color is at least $c$ are linked together by a series of *color-$c$ pointers*. We defer the implementation of color-$c$ pointers to later, but note here only that we can follow a pointer in $O(1)$ time.





Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

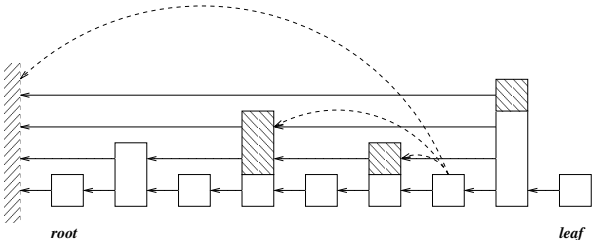



Figure 5: Diagram showing the colors assigned to a sequence of vertices with ranks 1 (root), 2, 1, 1, 1, 1, 1, 3, 1 (leaf). The unshaded portion of the tower of a vertex represents its rank; the shaded portion is the "additional" pointers added by the algorithm. Solid pointers show explicit color-$c$ pointers that would be stored in a biased skip list; dotted pointers shown are examples of pointers that are available implicitly through the *nca* operation.

The biased search starting at a node $v$ will proceed essentially as in a skip list. Let $c_v^{max}$ denote the maximum color of any ancestor of $v$, and $c^{max}$ the maximum color of any node in $H$. The search first tests $v$ – if the answer is "false" we are done, otherwise, we set $c = c_v^{max}$, and the current node to $u$, and suppose $v' = nca(v, c)$. We test at $v'$; if the outcome is "true" then we set the current node to $v'$; otherwise we check that $v'$ is not the final answer by testing the appropriate child of $v'$. If $v'$ is not the final answer then we set $c = c - 1$ and continue.

We now describe how we select the colors of the nodes in $T$. For any node $v$, denote the *rank* of $v$ to be $r_v = \lfloor \log_2 w_v \rfloor + 1$. We perform a pre-order traversal of each tree in $H$. When visiting $v$, we initially set $c_v = r_v$. Then, while the nearest ancestor of $v$ with color greater than or equal to $c_v$ has color exactly $c_v$, we increment $c_v$ by one (see Figure 1 for an example). We now show:

LEMMA 5.1. *(1) For $1 \le i \le c^{max} - 1$, between any two consecutive nodes of color $i$ there is a node of color $> i$; there is exactly one node of color $c^{max}$. (2) $c_v^{max} \le 1 + \log_2 W_v$; $c^{max} \le 1 + \log_2 N$. (3) For any vertex $v$ and ancestor $u$ of $v$, $c_v^{max} - c_u = O(\log(W_v/w_u))$.*

*Proof.* (1) follows by construction. For (2) and (3), consider any path in $H$ from a node $v$ to the root, and as in [10], define $N_i = |\{u \text{ an ancestor of } v : r_u = i\}|$ and $N_i' = |\{u \text{ an ancestor of } v : r_u \le i \text{ and } c_u \ge i\}|$. It is easy to see that:

$$N_{i+1}' \le N_{i+1} + \left\lfloor \frac{N_i'}{2} \right\rfloor \tag{5.1}$$

From this (2) and (3) follow as in [10]. □

From parts (1) and (3) of Lemma 5.1, it follows that a search that starts at a node $v$ and ends in a node $u$ takes $O(1 + \log(W_v/w_u))$ time. The following lemma shows that one can assign colors to all the nodes in $H$ in linear time.

LEMMA 5.2. *Given $H$ and the weights of the nodes, we can compute all node colors in $O(n)$ time.*

*Proof.* To assign the colors, keep a $c^{max}$-bit counter (which fits into one word); the counter is initialized to 0. We perform a pre-order traversal of $H$, and when we have visited a node $v$, the counter contains a 1 in bit position $i$ (the least significant bit is position 1 and the most significant is position $c^{max}$) if there is an ancestor of $v$ (including $v$ itself) with color $i$, such that there is no other node with color $> i$ between $v$ and this ancestor. Upon arriving at a node $v$ for the first time, we first compute $r_v$. Taking the value of the counter at $v$'s parent to be $x$, we set the lowest-order $r_v - 1$ bits of $x$ to 1, and add 1 to the result, giving a value $x'$. The counter value for $v$ is in fact $x'$, and is stored with $v$. To compute the color of $v$, we compute the bit-wise exclusive-OR of $x$ and $x'$, and find the position of the most significant 1 bit in the result. The implementation of the above in constant time requires standard $O(1)$-time bit-wise operations, most notably the $O(1)$-time computation of the MSB of a single word [23, 26]. □

**Nearest Colored Ancestor Problem.** We consider the following problem: Given a rooted ordered tree $T$ with $n$ nodes, each of which is assigned a color from $\{1, 2, \ldots, \sigma\}$, preprocess $T$ to answer the following query in $O(1)$ time:

$nca(v, c)$: given a node $v \in T$ and a color $c$, find the lowest ancestor of $v$ in $T$ whose color is $\ge c$.

We will use this data structure for every tree in $H$; clearly, the *nca* operation simulates following color-$c$ pointers, thus enabling biased search. To address our application, we consider the problem in the setting where word size $w$ is equal to the number of colors, $\sigma$. Our goal is to preprocess $T$ in $O(n)$ time, and store it in a data structure of size $O(n)$ words (i.e., $O(n\sigma)$ bits) to support in $O(1)$-time not only $nca()$ but also navigation queries, such as finding the distance between an ancestor and descendant, and choosing the $i$-th level-ancestor of a given node.

We partition the string BP of length $2n$ that stores the balanced parenthesis sequence of the given


Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

$n$-node tree into blocks of size $b = \min\{\sigma, \lg n\}$. Every node in the tree belongs to either one or two different blocks. For each block we identify a *representative* node which is the LCA of all the nodes whose corresponding parentheses are in that block. Thus there are $O(n/b)$ representative nodes. Our main idea is to preprocess each block so that queries whose answer lies within the block can be answered efficiently, as summarized in the following lemma. In addition, in linear time we compute and store all the answers for all the representative nodes.

LEMMA 5.3. *Given a block containing $b$ nodes where each node is associated with a color from the range $[1, \sigma]$, one can construct a $O(n \lg \sigma)$-bit structure in $o(b)$ time such nca queries whose answer lies within the same block can be answered in constant time.*

*Proof.* Our first step is to reduce the set of colors within a block from $\sigma$ to $O(b)$. (If $\sigma = b$, this step is omitted.) For each block, we obtain a sorted list of all colors that appear in that block. This can be done in linear time by sorting the pairs $\langle block_number, color_i \rangle$, where $color_i$ is the color of the $i$-th node (i.e., the node corresponding to the $i$-th parenthesis) in the block, using radix sort.

Let $c_1 < c_2 < \cdots < c_k$, for some $k \leq b$, be the set of all distinct colors that appear in a given block. Define $succ(c)$ to be the smallest $c_i$ such that $c_i \geq c$. Observe that $nca(x, c) = nca(x, succ(c))$, if the answer is within the block. For each block, we store the sorted sequence $c_1, c_2, \ldots, c_k$ of all distinct colors that appear in the block using an atomic heap [27], to support $succ()$ queries in constant time.

The range of colors in each block is now reduced to at most $b$. Thus, we need to answer the $nca()$ query in a block of size $b$ where the nodes are associated with colors in the range $[1, b]$. Using $b \lg b$ bits, we store the string consisting of the "reduced" colors of the nodes, in the same order as the nodes in the block. For each color $c$, $1 \leq c \leq b$, we build a $o(b)$-bit auxiliary structure that enables us to answer the query $nca(x, c)$ in constant time, for any node $x$ in the block if the answer lies within the block.

We divide each block (of size $b = \lg n$) into sub-blocks of size $s = \epsilon \lg n / \lg \lg n$, for some positive constant $\epsilon < 1$. If the answer to an $nca()$ query lies in the same sub-block as the query node, then we can find the answer using pre-computed tables, as all the information related to a sub-block (the parenthesis sequence and the 'reduced' color information of the nodes) fits in $O(\lg n)$ bits – the constant factor can be made less than $1/2$ by choosing the parameter $\epsilon$ in the sub-block size appropriately. If the answer to the query does not lie in the same sub-block, but with in the same block, then we first determine the sub-block (within the block) which contains the answer. To do this efficiently, we store the following additional information, for each block.

Given a reduced color $c$ in the block and a position $i$ within the block (corresponding to a node $x$), we define the *colored excess* of the position (with respect to the representative of the block) as the number of nodes with color $c$ in the path from $x$ to $rep(x)$. For every reduced color in the range $[1 \ldots b]$ and every sub-block, we compute and store the minimum and maximum colored excess values within the sub-block. Using this information for all the sub-blocks within a block, and for any particular color, we can find the sub-block containing the answer to a query with respect to that color (in constant time, using precomputed tables of negligible size). As there are $b/s$ sub-blocks and $b$ colors within each block, and the values stored for each sub-block are in the range $[0 \ldots b]$, the information stored for each block is $O((b/s)b \lg b) = O(b(\lg \lg n)^2)$ bits. Thus, over all the blocks, the space used is $O(n(\lg \lg n)^2)$ bits, which is $o(n)$ words. The computation of this information for all the sub-blocks can be performed in $O(n)$ time as explained below.

The total size of the information we need to store for each sub-block is $O(s(\lg \lg n)^2) = O(b \lg \lg n)$ bits, and we need to be able to read the information corresponding to all the sub-blocks within a block, corresponding to any particular color, by reading a constant number of $O(\lg n)$-bit "words". For this, we divide the range of colors (i.e., the range $[1 \ldots b]$) into chunks of size $d = s/\lg \lg n$, and write down the information corresponding to all the sub-blocks within a block, and of all the colors within a chunk, which fits in $O(\lg n)$ bits. Thus we can read the information corresponding to all the sub-blocks within a block, corresponding to any particular color, by reading these $O(\lg n)$-bits. We use precomputed tables to produce the information corresponding to each sub-block, and for all the colors within each chunk. Hence each sub-block has to be "processed" $O(b/d)$ times (as there are $b/d$ chunks). Thus the total time spent producing the information for all the sub-blocks and for all the chunks for each block is $O((b/s)(b/d)) = O(\lg \lg n)^3$. Thus the overall time spent for all the blocks is $O((n/b)(\lg \lg n)^3) = o(n)$. □

For each representative node $x$, we will store an array $A$ of size $\sigma$ such that $A_x[c] = nca(x, c)$, for $1 \leq c \leq \sigma$. As there are $O(n/b)$ representative





Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

nodes, and each entry in $A_x$ takes $\lg n$ bits, the total space used by arrays of all the representative nodes is $O((n/b)\sigma \lg n)$ bits which is $O(n\sigma)$. We will now describe how these arrays can be constructed with linear preprocessing time.

We first prove the following properties about the representative nodes.

LEMMA 5.4. *For each node $x$, at least one of these three statements is true: (i) $nca(x, c)$ lies in the (first) block to which $x$ belongs, (ii) $nca(x, c) = rep(x)$, or (iii) $nca(x, c) = nca(rep(x), c)$.*

*Proof.* The lemma follows from the following two observations:

- Either $nca(x, c) = parent(x)$, or $nca(x, c) = nca(parent(x), c)$.
- $rep(x)$ is either the highest ancestor of $x$ that is within the block containing $x$, or the lowest ancestor of $x$ that is outside the block containing $x$. (This follows from the fact that any block that contains nodes $x$ and $y$ also contains all the nodes along the path between $x$ and $y$ in the tree.) □

LEMMA 5.5. *Each representative node (except the root) has an ancestor within a height of at most $b$ from its level.*

*Proof.* Consider the lowest $b-1$ ancestors of a representative node $x$. Either the highest node, $y$, among these which is within the same block as $x$, or $y$'s parent, $z$ is a representative. Note that $y$ is the LCA of all nodes between $x$ and $y$, and if the block contains a sibling of $y$, then $z$ is the LCA of all nodes in the block. □

The root of the tree is a representative node, and the array for it consists of all null pointers. Traverse the tree in preorder, skipping all the non-representative nodes. When a representative node $x$ is reached, we will scan its ancestors starting from $x$ up to its lowest ancestor, $y$, that is also a representative. Let $A_y$ be the array stored at node $y$. During this upward scan, we will generate an array $B$ of length $\sigma$ as follows.

We keep track of the largest color value $c^{max}$ encountered at any point during the upward scan, and the first $c^{max}$ entries of the array $B$ are filled. In each step of the scan, if we encounter a node whose color value is at most $c^{max}$, we simply skip this node. On the other hand, if we encounter a node whose color value, $c$, is larger than $c^{max}$, then we set the entries $B[c^{max}+1], \ldots, B[c]$ to be pointers to the current node. We also update the value $c^{max}$ to be the new value $c$. We now copy $A_y$ to another array, and overwrite the first $c_{max}$ values of $A_y$ with the first $c^{max}$ values of $B$. The resulting array is the array $A_x$ that will be stored at node $x$. Generating the array $B$ takes $O(\lg n + b)$ time, as the length of $B$ is $O(\lg n)$, and it is "extended" at most $b$ times. Entries of $B$ are written using bit operations on words (note that the word size is $\sigma$). Thus the overall running time to generate all the arrays at the representative nodes is $O((n/b)(b + \lg n)) = O(n)$.

By plugging in this data structure in place of interval biased search trees, we get part(ii) of Theorems 1.1, 1.2 and 1.3.

## 6 Substring Decompression

We now extend our random access solutionsto efficiently support substring decompression. Note that we can always decompress a substring of length $m$ using $m$ random access computations. In this section we show how to do it using just 2 random access computations and additional $O(m)$ time. This immediately implies Theorem 1.1.

We extend the representation of $\mathcal{S}$ as follows. For each node $v$ in $\mathcal{S}$ we add a pointer to the next descendant node on the heavy path suffix for $v$ whose light child is to the left of the heavy path suffix and to the right of the heavy path suffix, respectively. This increases the space of the data structure by only a constant factor. Furthermore, we may compute these pointers during the construction of the heavy path decomposition of $\mathcal{S}$ without increasing the asymptotic complexity.

We decompress a substring $S[i, j]$ of length $m = j-i$ as follows. First, we compute the lowest common ancestor $v$ of the search paths for $i$ and $j$ by doing a top-down search for $i$ and $j$ in parallel. We then continue the search for $i$ and $j$ independently. Along each heavy-path on the search for $i$ we collect all subtrees to the left of the heavy path in a linked list using the above pointers. The concatenation of the linked list is the roots of subtrees to left of the search path from $v$ to $i$. Similarly, we compute the linked list of subtrees to the right of the search path from $v$ to $j$. Finally, we decode the subtrees from the linked lists thereby producing the string $S[i, j]$.

With our added pointers we construct the linked lists in time proportional to the length of the lists which is $O(m)$. Decoding each subtree uses time proportional to the size of the subtree. The total sizes of the subtrees is $O(m)$ and therefore decoding also takes $O(m)$ time. Adding the time for the two


Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

random access computations for $i$ and $j$ we obtain Theorem 1.1.

## 7 Compressed Approximate String Matching

We now show how to efficiently solve the compressed approximate string matching problem for grammar-compressed strings. Let $P$ and be string of length $m$ and let $k$ be an error threshold. We assume that the algorithms for the uncompressed problem produces the matches in sorted (as is the case for all solution that we are aware of). Otherwise, additional time for sorting should be included in the bounds.

To find all approximate occurrences of $P$ within $S$ without decompressing $\mathcal{S}$ we combine our substring decompression solution from the previous section with a technique for compressed approximate string matching on LZ78 and LZW compressed string [12].

We find the occurrences of $P$ in $S$ in a single bottom-up traversal of $\mathcal{S}$ using an algorithm for (uncompressed) approximate string matching as a black-box. At each node $v$ in $\mathcal{S}$ we compute the matches of $P$ in $S(v)$. If $v$ is a leaf we decompress the single character string $S(v)$ in constant time and run our approximate string matching algorithm. Otherwise, suppose that $v$ has left child $v_l$ and right child $v_r$. We have that $S(v) = S(v_l) \cdot S(v_r)$. We decompress the substring $S'$ of $S(v)$ consisting of the $\min\{|S(v_l)|, m+k\}$ last characters of $S(v_l)$ and the $\min\{|S(v_r)|, m+k\}$ first characters of $S(v_r)$ and run our approximate string matching algorithm on $P$ and $S'$. We compute the set of matches of $P$ in $S(v)$ by merging the list of matches from the matches of $P$ in $S(v_l)$, $S(v_r)$, $S'$ (we assume here that our approximate string matching algorithm produces list of matches in sorted order). This suffices since any approximate match with at most $k$ errors starting in $S(v_l)$ and ending in $S(v_r)$ must be contained within $S'$.

For each node $v$ in $\mathcal{S}$ we decompress a substring of length $O(m+k) = O(m)$, solve an approximate string matching problem between two strings of length $O(m)$, and merge lists of matches. Since there are $n$ nodes in $\mathcal{S}$ we do $n$ substrings decompression and approximate string matching computations on strings of length $m$ in total. The merging is done on disjoint matches in $S$ and therefore takes $O(\text{occ})$ time, where occ is the total number of matches of $P$ in $S$. With our substring decompression result from Theorem 1.1 and an arbitrary approximate string matching algorithm we obtain Theorem 1.2.

## 8 Random Access to Compressed Trees

Here we consider the problem of representing an ordered rooted tree $T$ (of arbitrary degree) with $N$ nodes, where each node is labelled with a character from $\Sigma$, where $T$ is compressed by sharing identical subtrees, giving a DAG with $n$ nodes (see Fig. 6).

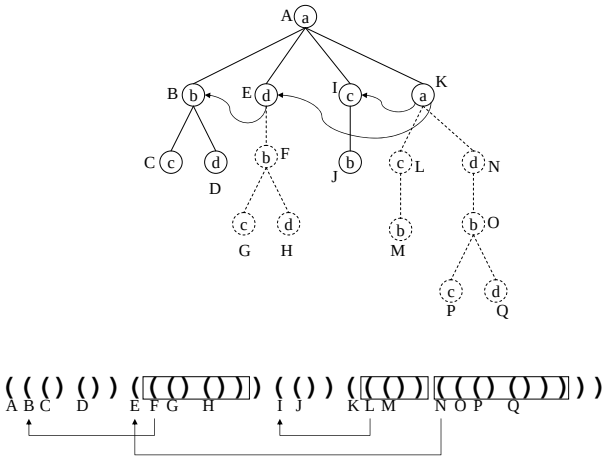



Figure 6: A compressed labelled tree given as a DAG and its balanced parentheses representation.

THEOREM 8.1. *Given labelled tree $T$ with $N$ nodes that is compressed to a rooted DAG $G$ with $m$ edges and $n$ nodes, there is a representation of $T$ that takes $O(m)$ words of space and supports the following navigational operations on $T$ in $O(\log N)$ time:*

- *select($x$): return the node with preorder number $x$,*
- *access($x$): return the label associated with node $x$,*
- *parent($x$): return the parent of node $x$,*
- *depth($x$): return the depth of node $x$,*
- *height($x$): return the height of node $x$,*
- *subtree_size($x$): return the size of the subtree rooted at $x$,*
- *$first_child(x)$: return the first child of $x$,*
- *next_sibling($x$): return the sibling immediately to the right of $x$,*





Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

- *level_ancestor*$(x, i)$: *return the ancestor* $x$ *at level* $i$*, and*
- $LCA(x)$*: return the lowest common ancestor of the nodes* $x$ *and* $y$*.*

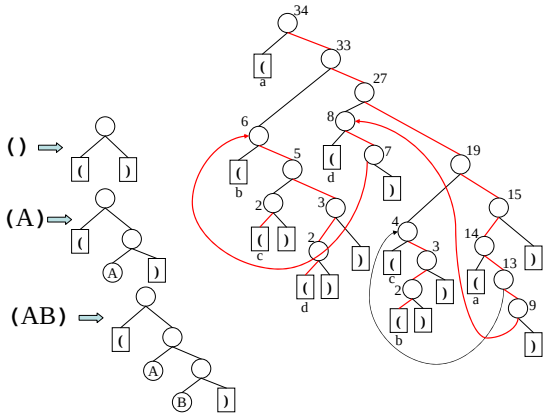



Figure 7: The SLP representing the balanced parenthesis string of the tree in Fig. 6 – numbers above an internal node (non-terminal) represent the lengths of strings output by that non-terminal. Heavy paths are shown by red arrows – there are three in all.

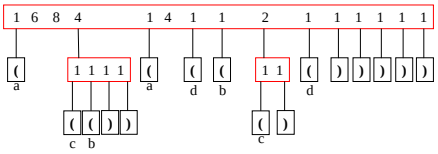



Figure 8: The left and right size sequences for the three heavy paths in the SLP of Fig. 7.

*Proof.* First we create an SLP representing the balanced parentheses representation [46] of the tree $T$

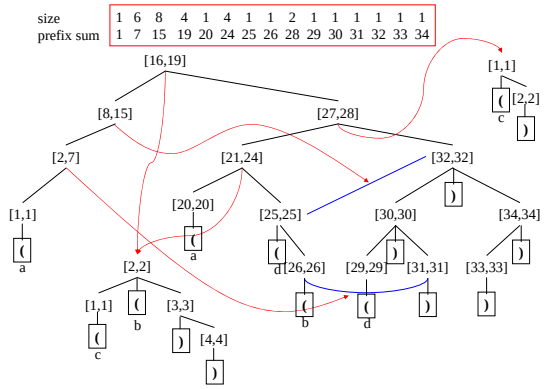



Figure 9: The interval-biased search trees. Each internal nodes have left and right children, shown in black. The "middle" child pointer (in red) is either a pointer to a terminal node, a pointer to another node, or a pair of pointers to leaves (pair indicated in blue).

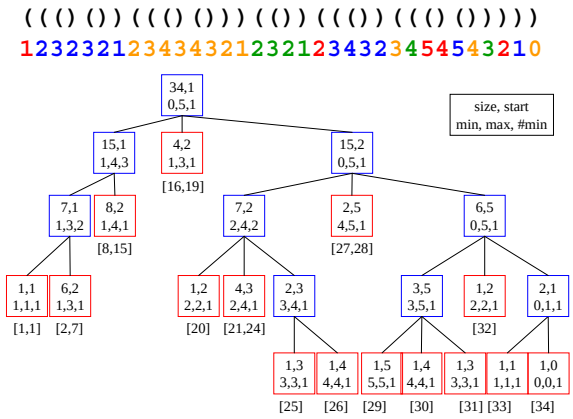



Figure 10: The range min-max tree of the tree. Colours in the "excess" values correspond to the subranges of the parenthesis bit-string represented by leaves of the min-max tree.

as follows. For each node $x$ of the DAG $G$ with $k \geq 0$ children, we create $k + 1$ non-terminal nodes $x_0, \ldots, x_k$. The node $x_0$ and $x_k$ has '(' and ')' as its left and right child, respectively. The node $x_i$ $(0 \leq i \leq k - 1)$ has $x_{i+1}$ as its right child, and the node $x_j$ $(1 \leq j \leq k)$ has the corresponding represen-



Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

tation of the $i$-th child of the node $x$ of $G$ (see Fig. 7). Clearly, the size of this SLP is $O(m)$.

Next we apply the heavy path decomposition for the SLP of $T$ based on the size of the string generated by each nonterminal. For each heavy path, we make the left and right size sequences and concatenate them into one, reversing the right size sequence (see Fig. 8). By doing so, the numbers in the combined size sequences represent consecutive sub-strings of the balanced parenthesis string. Then we construct the interval-biased search tree for each heavy path, and associate each each node of the interval-biased search tree the substring generated by the light child it represents; we store with each node the interval $[s, t] \subset [1, 2N]$ of positions in the balanced parentheses sequence spanned by that light child. Each node in the interval search tree has (possibly null) left and right children, and also has an extra "middle" child, which represents the balanced parentheses sequence associated with that interval search tree node. The "middle" child is either a leaf, which corresponds to a single character in the sequence, a pointer to another node of the SLP, or a pair of pointers to two leaves of the SLP. A pair of pointers to the leaf $a$ at position $p_a$ and the leaf $b$ at position $p_b$ represents the interval $[p_a, p_b] \subset [1, 2n]$ of the balanced parentheses sequence.

By using this data structure, any interval of length $x$ can be represented by a subtree of height at most $1 + \log x$, or two subtrees of height at most $1 + \log x$, and the lowest common ancestor of the roots of the subtrees. Therefore an interval-biased search for finding an interval of length $y$ takes $O(\log(x/y))$ time. This means that the $i$-th character of the balanced parenthesis sequence is obtained in $O(\log N)$ time. Similarly, by using the range min-max tree [53], we can perform all the desired tree operations in $O(\log N)$ time. To accomplish this, we store, in each node of the SLP, the subtree size, minimum, maximum, and start value of the excess values in the subtree, and the number of minimum excess values in the subtree (see Fig. 10).

## References


[1] N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical report, TR-71/87, Institute of Computer Science, Tel Aviv University, 1987.

[2] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proceedings of the 39th annual symposium on Foundations Of Computer Science (FOCS)*, pages 534–543, 1998.

[3] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Comp. and Sys. Sciences*, 52(2):299–307, 1996.

[4] A. Amir, G. Landau, and D. Sokol. Inplace 2d matching in compressed images. In *Proc. of the 14th annual ACM-SIAM Symposium On Discrete Algorithms, (SODA)*, pages 853–862, 2003.

[5] A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004. Announced at SODA 2000.

[6] A. Apostolico and S. Lonardi. Some theory and practice of greedy off-line textual substitution. In *Proc. IEEE Data compression conference*, pages 119–128, 1998.

[7] A. Apostolico and S. Lonardi. Compression of biological sequences by greedy off-line textual substitution. In *Proc. IEEE Data compression conference*, pages 143–152, 2000.

[8] A. Apostolico and S. Lonardi. Off-line compression by greedy textual substitution. *Proc. IEEE*, 88(11):1733–1744, 2000.

[9] O. Arbell, G. M. Landau, and J. Mitchell. Edit distance of run-length encoded strings. *Information Processing Letters*, 83(6):307–314, 2001.

[10] A. Bagchi, A. L. Buchsbaum, and M. Goodrich. Biased skip lists. *Algorithmica*, 42:31–48, 2005.

[11] S. W. Bent, D. D. Sleator, and R. E. Tarjan. Biased search trees. *SIAM J. Comput.*, 14(3):545–568, 1985.

[12] P. Bille, R. Fagerberg, and I. L. Gørtz. Improved approximate string matching and regular expression matching on ziv-lempel compressed texts. *ACM Transactions on Algorithms*. To appear. Announced at CPM 2007.

[13] P. Buneman, B. Choi, W. Fan, R. Hutchison, R. Mann, and S. Viglas. Vectorizing and querying large xml repositories. In *ICDE*, pages 261–272. IEEE Computer Society, 2005.

[14] H. Bunke and J. Csirik. An improved algorithm for computing the edit distance of run length coded strings. *Information Processing Letters*, 54:93–96, 1995.

[15] G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of xml document trees. *Inf. Syst.*, 33(4-5):456–474, 2008.

[16] P. Cégielski, I. Guessarian, Y. Lifshits, and Y. Matiyasevich. Window subsequence problems for compressed texts. In *Proc. of the 1st symp. on Computer Science in Russia (CSR)*, pages 127–136, 2006.

[17] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005. Announced at STOC 2002 and SODA 2002.

[18] B. Chazelle and B. Rosenberg. Computing partial






Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

sums in multidimensional arrays. In *Proceedings of the 5th annual ACM Symposium on Computational Geometry (SCG)*, pages 131–139, 1989.

[19] F. Claude and G. Navarro. Self-indexed text compression using straight-line programs. In *Proc. 34th Mathematical Foundations of Computer Science*, volume 5734 of *Lecture Notes in Computer Science*, pages 235–246, 2009.

[20] R. Cole and R. Hariharan. Approximate string matching: A simpler faster algorithm. *SIAM J. Comput.*, 31(6):1761–1782, 2002.

[21] M. Crochemore, G. Landau, and M. Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM Journal on Computing*, 32:1654–1673, 2003.

[22] O. Delpratt, R. Raman, and N. Rahman. Engineering succinct dom. In A. Kemper, P. Valduriez, N. Mouaddib, J. Teubner, M. Bouzeghoub, V. Markl, L. Amsaleg, and I. Manolescu, editors, *EDBT*, volume 261 of *ACM International Conference Proceeding Series*, pages 49–60. ACM, 2008.

[23] F. Ellen. Constant-time operations for words of length $w$. 1999.

[24] M. Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *Proceedings of the 7th Symposium on Combinatorial Pattern Matching (CPM)*, pages 130–140. Springer, 1996.

[25] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1), 2009.

[26] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

[27] M. L. Fredman and D. E. Willard. Transdichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, 1994.

[28] P. Gage. A new algorithm for data compression. *The C Users J.*, 12(2):23 – 38, 1994.

[29] L. Gasieniec, R. Kolpakov, I. Potapov, and P. Sant. Real-time traversal in grammar-based compressed files. In *Proceedings of the Data Compression Conference*, pages 458–458, 2005.

[30] L. Gasieniec and I. Potapov. Time/space efficient compressed pattern matching. *Fundam. Inf.*, 56(1,2):137–154, 2003.

[31] T. Hagerup. Sorting and searching on the word ram. In M. Morvan, C. Meinel, and D. Krob, editors, *STACS*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer, 1998.

[32] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[33] D. Hermelin, S. Landau, G. Landau, and O. Weimann. A unified algorithm for accelerating edit-distance via text-compression. In *Proc. of the 26th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 529–540, 2009.

[34] J. Iacono. private communication. 2010.

[35] J. Karkkainen, G. Navarro, and E. Ukkonen. Approximate string matching over Ziv-Lempel compressed text. In *Proc. of the 11th symposium on Combinatorial Pattern Matching (CPM)*, pages 195–209, 2000.

[36] J. Karkkainen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. of the 3rd South American Workshop on String Processing (WSP)*, pages 141–155, 1996.

[37] J. C. Kieffer and E. H. Yang. Grammar based codes: A new class of universal lossless source codes. *IEEE Trans. Inf. Theory*, 46(3):737–754, 2000.

[38] J. C. Kieffer, E. H. Yang, G. J. Nelson, and P. Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Trans. Inf. Theory*, 46(5):1227 – 1245, 2000.

[39] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

[40] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *J. Algorithms*, 10(2):157–169, 1989.

[41] J. N. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. IEEE*, 88(11):1722 – 1732, 2000. Announced at DCC 1999.

[42] Y. Lifshits. Processing compressed texts: A tractability border. In *Proc. of the 18th symposium on Combinatorial Pattern Matching (CPM)*, pages 228–240, 2007.

[43] V. Makinen, G. Navarro, and E. Ukkonen. Approximate matching of run-length compressed strings. In *Proc. of the 12th Symposium On Combinatorial Pattern Matching (CPM)*, pages 1–13, 1999.

[44] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. In *Proc of the 5th Symposium On Combinatorial Pattern Matching (CPM)*, pages 31–49, 1994.

[45] K. Mehlhorn. Nearly optimal binary search trees. *Acta Informatica*, 5:287–295, 1975.

[46] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[47] J. I. Munro and S. S. Rao. Succinct representations of functions. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 1006–1015, 2004.

[48] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[49] G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Faster approximate string matching over compressed text. In *Proc. of the 11th Data Compression Conference (DCC)*, pages 459–468, 2001.

[50] C. G. Nevill-Manning and I. H. Witten. Identifying



Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

hierarchical strcture in sequences: A linear-time algorithm. *J. Artif. Intell. Res. (JAIR)*, 7:67–82, 1997.
[51] M. Pătraşcu. private communication. 2009.
[52] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.
[53] K. Sadakane and G. Navarro. Fully-Functional Succinct Trees. In *Proc. ACM-SIAM SODA*, pages 134–149, Jan. 2010.
[54] P. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms*, 1(4):359–373, 1980.
[55] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Byte Pair encoding: A text compression scheme that accelerates pattern matching. *Technical Report DOI-TR-161, Department of Informatics, Kyushu University*, 1999.
[56] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Speeding up pattern matching by text compression. In *Proc. of the 4th Italian Conference Algorithms and Complexity (CIAC)*, pages 306–315, 2000.
[57] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.
[58] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.
[59] E. H. Yang and J. C. Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform – part one: Without context models. *IEEE Trans. Inf. Theory*, 46(3):755–754, 2000.
[60] J. Ziv and A. Lempel. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.
[61] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.



Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.